Fun with Tuples

Marshall Clow
Qualcomm
mclow.lists@gmail.com
Twitter: mclow

C++Now - May 2013

What do I mean, fun?

- Basic information about tuples
- What you can do with them
- Interesting techniques that I've found/discovered
- No emphasis on usability or practicality in this talk.

So, what's a tuple?

- std::tuple introduced in C++2011
- A generalization of std::pair.
  - Arbitrary number of elements
- No names for the fields
  - Sadly. (first, second, third, nineteenth, fivehundredthirtyseventh)

What’s the difference between a tuple and a struct?

- Field names
- Layout
-

What can you do with a tuple?

- std::get<N> (tuple) -- constexpr
- std::tuple_element<N> (tuple)::type -- constexpr
- std::tuple_size (tuple) -- constexpr
- compare them (==, !=, <, etc)

How do I make a tuple?

- typedef tuple<int, float, string> Tuple;
- Tuple t1 { 3, 2.78, “Hi Mom” };
- Tuple t2 = make_tuple (3, 2.78f, string(“Hi Mom”));
- Tuple t3 = tuple_cat (make_pair (3, 2.78f), make_tuple(string(“Hi Mom”)));

std::tie

- Creates a tuple of lvalue references
- Useful for bursting a tuple into a sequence of variables
- Makes functions that return multiple values easy to use
- std::ignore useful for saying “I don’t want this value”
- “An object of unspecified type such that any value can be assigned to it with no effect”

```
int main ( int, char ** ) {
    using namespace std;
    auto tup = make_tuple ( 3, 3.14, string ("Hi Mom" ));
    int i;
    tie ( i, ignore, ignore ) = tup; // i is now 3

    // Fun with ignore
    ignore = 4;
    ignore = tup;
    ignore = string ("Hi Mom" );
    ignore = ignore;
    auto devNull = ignore;
    devNull = tup;
    return 0;
    }
```

Comparing Tuples

- operator == is defined as
  - get<1>(t1) == get<1>(t2) && get<2>(t1) == get<2>(t2) ...
- the relational operators are defined as a lexicographic compare

```
struct S {
  int i;
  float f;
  string s;
  };

  S one { 4, 3.2f, “Hi” };
  S two { 4, 3.2f, “Mom” };

  tie ( one.i, one.f, one.s ) ==
       tie ( two.i, two.f, two.s );
  tie ( one.i, one.f) == tie ( two.i, two.f );

  tie ( one.i, one.f, one.s ) <
       tie ( two.i, two.f, two.s );
  tie ( one.s, one.f, one.i ) <
       tie ( two.s, two.f, two.i );
```

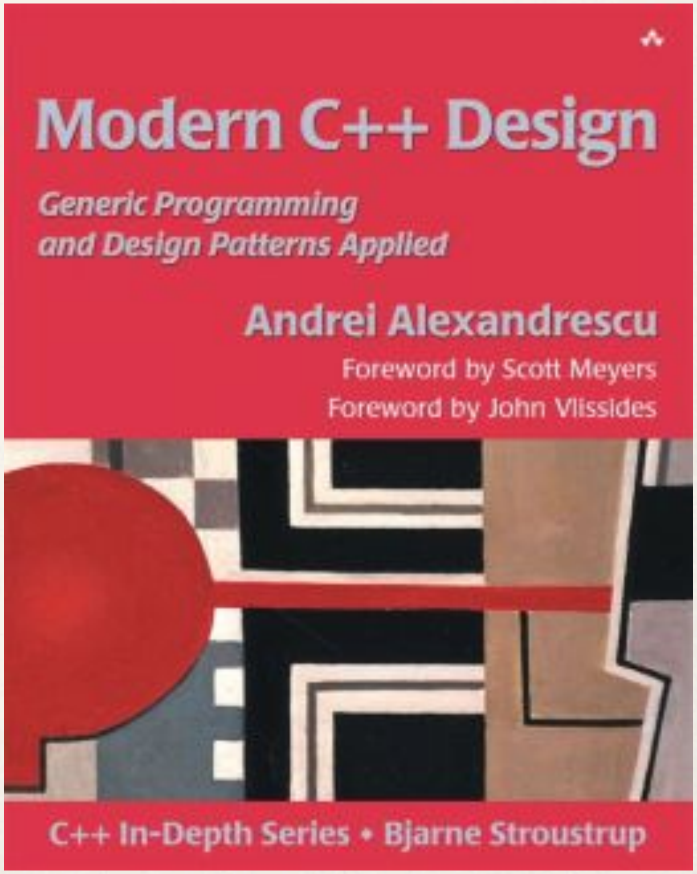
Is std::tuple a container?

- Not like vector/list/etc, because the elements can be heterogeneous.
- But at compile time...

A container of types

- std::tuple_element<N>(t)::type -- returns the type of the Nth element of the tuple.
- Consider "typedef std::tuple<int, const char*, void> Tuple;"
  - Is this legal?

Who remembers this book?


Modern C++ Design
Generic Programming
and Design Patterns Applied
Andrei Alexandrescu
Foreword by Scott Meyers
Foreword by John Vlissides
C++ In-Depth Series • Bjarne Stroustrup

Tuples and variadic templates

- Tuples are implemented as variadic templates

- Variadic templates are **the** tool for manipulating tuples

Printing a tuple

```
//   Print a tuple
//   Based on http://cpplove.blogspot.com/2012/07/printing-tuples.html
     template<std::size_t> struct int_{};

//   Forward declaration
     template <typename... Args>
     std::ostream& operator<<(std::ostream& out, const std::tuple<Args...>& t);

//   Deal with pair, too
     template <typename T1, typename T2>
     std::ostream& operator<<(std::ostream& out, const std::pair<T1, T2>& p) {
         return out << '(' << p.first << ", " << p.second << ')';
         }

     template <typename Tuple, size_t Pos>
     std::ostream& print_tuple(std::ostream& out, const Tuple& t, int_<Pos> ) {
       out << std::get< std::tuple_size<Tuple>::value-Pos>(t) << ", ";
       return print_tuple(out, t, int_<Pos-1>());
     }

     template <typename Tuple>
     std::ostream& print_tuple(std::ostream& out, const Tuple& t, int_<1> ) {
       return out << std::get<std::tuple_size<Tuple>::value-1>(t);
     }

     template <typename... Args>
     std::ostream& operator<<(std::ostream& out, const std::tuple<Args...>& t) {
       out << '(';
       print_tuple(out, t, int_<sizeof...(Args)>());
       return out << ')';
     }
```

```
int main ( int, char ** ) {
    std::tuple<int, std::string, float> t1
                                    {10, "Test", 3.14};
    std::cout << "t1:" << t1 << std::endl;

    std::tuple<int, std::tuple<std::string, float>> t2
                    { 10, std::make_tuple ("Test", 3.14 )};
    std::cout << "t2:" << t2 << std::endl;

   auto t3 = std::make_tuple ( t1,
      std::make_pair ( "Foo",
         std::make_tuple ( "Nest", 23, 2.71, "bar")), t1 );
   std::cout << "t3:" << t3 << std::endl;

   return 0;
   }
```

Sequences of integers

- When you are picking out elements of a tuple, you need an index.
- Usually more than one.
- Enter:
    - template <size_t... Idx> struct indices {};

```
//Select a subset of a tuple at run time
template <typename ...Ts, size_t ...Is>
auto
select(tuple<Ts...> t, indices<Is...>) ->
   decltype(make_tuple( get<Is>(t)... ))
{
   return make_tuple( get<Is>(t)... );
}

//Select a subset of a tuple at compile time
template <typename Tuple, size_t ...Is>
struct select_ {
   typedef
decltype(make_tuple(get<Is>(Tuple())... )) type;
   };
```

```
std::tuple<int, std::string, float> t1 {10, "Test", 3.14};

// Make a new tuple with the old values
auto t4 = select ( t1, indices<0,2,1>());

// Make a new tuple type and instantiate it
typedef select_<std::tuple<int, float, std::string>, 0,2,1>::type T5;
T5 t5 = std::make_tuple ( 3, "Hi Mom", 3.14 );
```

What can we do with this?

- Pretty much any transformation of a tuple (or a tuple type).
- The transformation has to be determined at compile-time.

Apply

- Take a functor and a tuple of values.
- Call the functor with the elements of the tuple as parameters.

```
template<typename F, typename Tuple, int... I>
auto
apply(F&& f, Tuple&& args, indicies<I...>) ->
  decltype(forward<F>(f)(get<I>(forward<Tuple>(args))...))
  {
    return forward<F>(f)(get<I>(forward<Tuple>(args))...);
  }
```

More calling tricks

- Given a collection of functors and a collection of tuples, call each functor on the associated tuple, and return a tuple of values
  - Each on their own thread
- Apply a functor to each element in a tuple, and return the results as a tuple
  - Wrap the tuple value in boost::any?

Conclusions

- For such a simple data structure, there’s a lot to be done with tuple
- I’m pretty sure that I’ve just scratched the surface here
- Go out and have fun with tuples!

Questions?